プログラミング 2−19
（1995. 8. 25）

メタレベル計算を用いた協調処理の実現

山崎 賢治, 楢崎 修二, 牛島 和夫
yamaken@csce.kyushu-u.ac.jp
九州大学 工学部
〒812-81 九州大学工学部情報工学科

分散人工知能の研究においてプログラムの効率よい実行のための協調処理を実現する様々なアルゴリズムが提案されてきた。しかしそのようなアルゴリズムを具体的な分散問題解決プログラムの中で利用するための枠組については研究がなされていない。そこで我々はメタレベル計算としての協調処理の記述方法を提案し、その枠組に基づいたシステムを、CLOS MOPを用いて実装する。本稿では、提案する記述方法と、その枠組に基づいたシステムの実装とについて説明する。さらにエージェント全体の知識の一貫性保持に関する協調処理2つを例にとり、プログラムの簡潔さについて考察し、提案した記述方法の評価を行なう。

Implementation of cooperative processing with metalevel computation

Kenji Yamasaki, Shuji Narazaki, Kazuo Ushijima
Faculty of Engineering, Kyushu University
Dept. of Comp. Sci. and Comm. Eng., Kyushu University, 812-81 Japan

For the efficient execution of distributed programs, many algorithms for cooperative processing have been proposed in the area of Distributed Artificial Intelligence(DAI). But few research has been done about framework for using algorithms in practical problem solving programs. Thus we propose a way of description of cooperative processing as metalevel computation and implement a system based on our framework with CLOS MOP. In this paper, we explain the way of description scheme and the implementation of system. And through two examples of cooperative processing about coherency management of shared knowledge, we evaluate the proposed framework in terms of simplicity of the programs.

1 はじめに

分散問題解決は、複数のエージェントと呼ばれる知的システムが一つの問題を部分問題に分割して効率よく問題を解決する手法である。分散問題解決を用いることによって、並列探索などの問題は効率よく解決できる [2]。エージェントに対してどのように問題を分割するかは問題解決アルゴリズムによって決まる。

エージェントは目標を持ち、実行環境 (問題の解決状況、計算資源の量など) の動的な変化に応じて各自で自分の動作を動的に決定することができる。各エージェントは、自分の動作計画に影響するエージェントの全体集合または部分集合との通信によって実行環境の情報を獲得する。これらの性質を生かして各エージェントが、実行環境の情報をもとに全体の処理効率を考慮して動作すれば、問題は効率よく解決できる。このような処理をエージェントの協調処理と呼ぶ。協調処理を用いなくても問題は解決できる。しかし動的な協調処理を行なえばより効率よく問題を解決できることが期待できる。つまりここで言う協調とは、問題解決プログラムが与えられた時にその中で効率よく問題を解くための協調である。

分散問題解決、より一般的には分散人工知能の研究において、協調処理を実現した様々なアルゴリズム (以下、協調戦略と呼ぶ) が提案されている [2, 3, 5, 11]。しかし、そのようなアルゴリズムを具体的な分散問題解決プログラムの中で利用するための枠組については研究がなされていない。問題解決アルゴリズムは問題ごとに異なる。しかしエージェントの協調戦略は様々な分散問題解決に利用できるので、共通性が高いと言える。しかし現状の記述方法では、問題解決アルゴリズムと協調戦略とが明確に分けて考えられていないので協調戦略の再利用が困難になっている。

本研究の目的は、このような問題を解消するような記述方法を、分散問題解決プログラムを記述するプログラマにとって有効な形で提案することである。そこで我々は、メタレベル計算を用いた協調処理の実現方法を提案し、実際に CLOS(Common Lisp Object System)[9] MOP(Metaobject Protocol)[4, 8] を用いて、提案した記述方法に基づいたシステムの実装を行なった。さらにエージェント全体の知識の一貫性保持に関する協調処理2つを例にとり、プログラムの簡潔さについて考察し、提案した記述方法の評価を行なった。

本稿は以下のような構成になっている。まず第2章でメタレベル計算を用いた協調処理の実現方針について説明し、第3章で実装に用いる CLOS MOP について説明する。次に第4章では CLOS MOP を用いた、提案した記述方法に基づいたシステムの実装について具体的に解説し、第5章で2つの具体的な協調戦略に対して、実装したシステムの利用例を示し、その評価を行なう。最後に第6章でまとめ、今後の課題を述べる。

2 協調戦略の分離記述について

本章ではエージェントの協調戦略についてより詳しく考察し、協調戦略の再利用を可能にするような記述方法の提案を行なう。

図 1はエージェントの協調戦略の概要を示した図である。まずエージェントは協調のための動作決定に必要な、実行環境に関する情報を作成する。これを実行環境モデルと呼ぶ。実行環境モデルは、局所情報から推測して作成する場合と、他のエージェントから通信によって獲得した情報を基にして作成する場合がある。エージェントはこの実行環境モデルを評価することによって全体の処理効率を上げるような動作を決定する。つまり動作決定の必要があるのはエージェントの内部状態が変化した時である。従ってエージェントの協調戦略は、局所情報の更新に伴って起動すべきものである。以上の考察により、協調戦略は一種の自己反映的計算であるといえる。ここで自己反映的計算 (**computational reflection**)[7] とは、自分の実行状況を基に自分自身の行動を決定するという計算である。

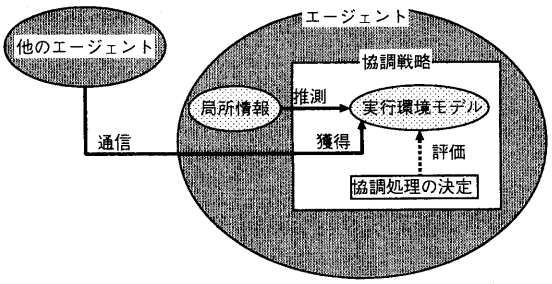

図 1: エージェントの協調戦略

そこで本研究では、問題解決アルゴリズムを記述したプログラム (以下、問題レベルのプログラムと呼ぶ) から協調戦略を分離して記述するという記述方法を提案する (図 2)。この記述方法の狙いは、共通性の高い協調戦略を分離記述することによって、多様な問題解決プログラムに対しての再利用を可能にすることである。

ところで、分離記述した協調戦略の再利用を実現するための方法としては以下のようなものが考えられる。

(1) 協調処理を、問題レベルのプログラムで呼び出す関数手続きとして実現する。

(2) 協調処理を、メタレベルの計算として実現する。

ここで、(1) の方法をとった場合には局所情報が変化するたびに協調戦略の関数呼び出しを行なわねばならな

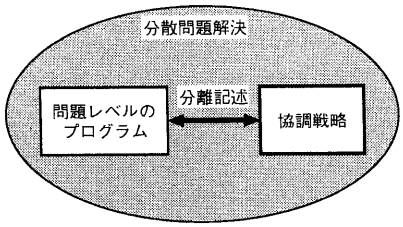

図 2: 協調戦略の分離記述

い。これに対して (2) の方法をとった場合には、協調戦略がエージェント自身の持つ局所情報の変化に付随するメタレベル計算として起動される。これによりエージェントの協調戦略に関する処理を自動化することができるので、協調戦略に関する命令を問題レベルのプログラムに記述する必要がなくなる。従って問題レベルのプログラムの保守性が向上する。(2) の方法を用いた方がよいのは明白である。

3 システムの実装手段について

前章で我々は協調戦略をメタレベル計算として分離記述するという記述方針を提案した。この方法で協調戦略を利用したプログラムを記述できるようにするためには、その枠組に基づいたシステムを実装することがまず必要になる。そこで本研究では、MOP(Metaobject Protocol)[4, 8] を持つ言語で実装を行なうことにした。MOP を持つ言語を用いれば、必要に応じてその言語の意味論を拡張することができるからである。本研究では MOP を持つ CLOS(Common Lisp Object System)[9] を用いた。簡単に言えば MOP を用いて CLOS という言語を、協調処理をメタレベル計算として実現できるような言語に拡張するということになる。特に CLOS の MOP を CLOS MOP と呼んでいる。使用した処理系である CMUCL(Carnegie Mellon University Common Lisp) は CLOS の実装を 2 つ持っている。本研究では MOP を利用するために、そのうちの 1 つである PCL (Portable Common Loops) の実装を用いた。

この章では具体的な実装方針について述べる準備として、CLOS MOP についての簡単な説明を行なう [4]。

CLOS の設計における静的な部分をメタレベルアーキテクチャと呼び、CLOS の構成要素がどのように CLOS を構成しているのかを記述している。構成要素とは、クラス、メソッド、そしてスロットなどである。これらの構成要素はメタレベルでのクラスの継承木として表現される (図 3)。図 3において最も深い部分に位置する 3 つのメタクラスはシステムの実装の際に追加定義した要素である。これらについては 4 章で述べることにする。

動的な部分はプロトコルという表現で記述され、実行時の言語の振舞いに影響するような動作をする構成要素の操作を規定している。言語の振舞いに対して、どのように構成要素が変化し、どのように相互作用し合うかを 1 つもしくはそれ以上のプロトコルによって特定している。

CLOS という言語を、言語自身と、その概念を抽象的に記述して実装しているメタレベルというものとに分けた場合、このメタレベルという世界が集合的に CLOS MOP として知られるようになったものである。

伝統的に言語の設計において、その意味論や実装はブラックボックス化され、言語の利用者がこれらを管理することはできなかった。一方 CLOS MOP は、CLOS の抽象概念やその実装を言語の利用者に公開している。つまり言語の利用者は CLOS のコードを用いて CLOS のメタな部分を参照できる。さらにメタクラスの追加定義や新しいメソッドの定義などを用いてメタな部分を付加的に拡張することによって、元来 CLOS に内在する実装コードやプログラムなどに影響を及ぼすことなしに CLOS という言語自体を拡張できる。例えばインスタンスの表現に関する実装方法や、多重継承の振舞いに関する言語意味論などを必要に応じて調節することができるのである。しかし CLOS MOP は CLOS の実装をすべて公開しているわけではない。実装の本質的構造のみである。つまり言語の利用者が手を加えることのできる許容範囲は、MOP が規定しているのである。

4 システムの実装について

本章では分離記述した協調戦略を、問題レベルプログラムに対するメタレベル計算として利用できるようなシステムの実装について説明する。

協調処理がエージェント自身の持つ局所情報の更新または参照に付随するメタレベル計算として起動すべき処理であることはすでに述べた。そこでまず本研究では、「各エージェント自身が持つ 1 つの局所変数の更新または参照に付随して起動すべき協調戦略」をメタレベル計算として実現するためのシステムを、CLOS MOP を用いて実装した。この実装方法について説明するために、まず以上の概念が CLOS プログラムとどのように対応するのかを以下に示す。

- エージェント ↔ 問題レベルプログラムで定義するエージェントクラスのインスタンス
- エージェントの持つ局所情報 ↔ エージェントクラスのスロット
- 局所情報の更新、参照 ↔ スロット値の更新、参照

このシステムを用いれば 1 つのスロットの更新と参照とにそれぞれ 1 つの協調戦略を対応させて自動的に起動

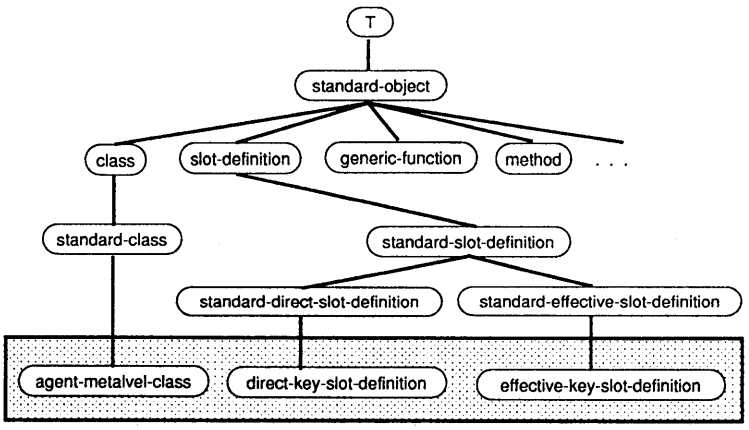

図 3: メタオブジェクトのクラス階層

させることができる。協調戦略を起動する契機となるスロットを以下ではキースロットと呼ぶことにする。

4.1 MOP を用いた処理系の拡張

ここではシステムの実装のために MOP を用いて CLOS のどの部分を拡張したかについて順に説明してゆく。MOP を用いた処理系の拡張は、まず新しいメタクラスを定義し、次にそのメタクラスに対して特定化 (specified) したメソッドを既存の総称関数に追加するという手順で行なう。

まず以下のようなメタクラスを新しく定義した。

agent-metalevel-class

この新しいメタクラスは standard-class という既存のメタクラスのサブクラスとして定義した。standard-class とは CLOS の利用者が defclass を用いて定義したクラスの、デフォルトの振舞いを司るメタクラスである。ここで定義した agent-metalevel-class をクラス定義の際にメタクラスとして指定するには:metaclass というクラスオプションを用いればよい。

direct-key-slot-definition,effective-key-slot-definition

これらの新しいメタクラスは standard-direct (effective)-slot-definition という既存のメタクラスのサブクラスとして定義した。standard-direct(effective)-slot-definition とは CLOS の利用者がクラス定義の際に定義したスロットクラスの、デフォルトの振る舞いを司るメタクラスである。ここで定義したメタクラスは、協調戦略を起動する契機となるスロット (キースロット) のメタクラスとして用いる。さらにこのメタクラス定義では、新しいスロットオプション (後述) を 2 つ定義している。

メタレベル計算は、総称関数が呼び出された時の内部計算であると言い換えることができる。総称関数は引数の一つとしてメタクラスをとる。総称関数はそのメタクラスに特定化されたメソッドを選択し、実行する。従って、上記の新しいメタクラスに特定化したメソッドを定義することにより、新しい振舞いを与えることができる。つまり、協調戦略をその中で呼び出せばメタレベル計算として協調処理を実現できることになる。以下では新しく定義した主要なメソッドについて説明してゆく。

direct(effective)-slot-definition-class

これらはクラスの初期化においてスロットクラスのメタクラスを決定する総称関数である。新しく定義したスロットオプションを用いて定義してあるスロット (つまりキースロットとして定義してあるスロット) に対しては、そのメタクラスを direct(effective)-key-slot-definition に設定するようにした。

(setf slot-value-using-class)

スロットへの代入の際に呼び出される総称関数である。スロットオプション:strategy-set(後述) を用いて定義されたキースロットへの代入が起こった場合に、まずキースロットの実体 (メソッド%instance-ref によって参照されるオブジェクト) を cons に初期化し、car には

スロット値を、cdrには実行環境モデルを割り当てるための領域とする(図4)。そしてその次にcdrを引数として、指定された協調戦略を呼び出すようにした。

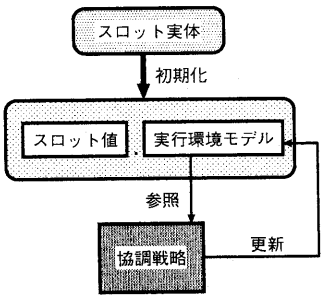

図 4: キースロットの実体の初期化

slot-value-using-class

スロット値の参照の際に呼び出される総称関数である。スロットオプション:strategy-ref(後述)を用いて定義されたキースロットの参照が起こった場合に、上と同様にキースロットの実体を初期化し、指定された協調戦略を呼び出すようにした。

4.2 利用者インターフェースについて

本研究で実装したシステムの概要をまとめたものが図5である。ここでは、このシステムを利用して分散問題解決プログラムを記述するプログラマに対して提供する利用者インターフェースについて説明する。利用者に対して提供しているのは、エージェントクラスの定義とエージェント間の通信とに関する部分、そして分離記述する協調戦略の記述方法である。

defagent マクロ

エージェントクラスを定義する際に用いるマクロである。このマクロはクラス定義マクロdefclassを拡張したものである。このマクロを用いてエージェントクラスを定義することにより、そのクラスのメタクラスが自動的にagent-metalevel-classに設定される。また、自動的にidスロットが定義されるので、各エージェントをid番号(0,1...)で区別するのに用いることができる。さらにこのマクロを用いた定義の中では以下に挙げる2つの新しいスロットオプションを用いることができる。

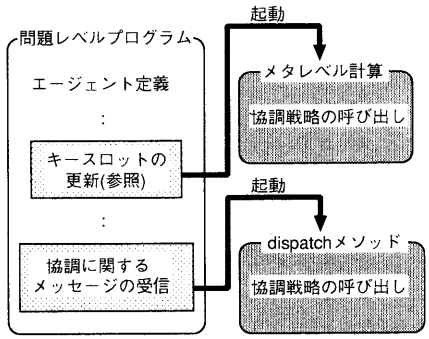

図 5: システムの概要

:strategy-set *function*

:strategy-ref *function*

前者は代入が起こった時に協調処理を起動するキースロット、後者は参照が起こった時に協調処理を起動するキースロットを定義する際にそれぞれ用いる。これらのスロットオプションを用いて指定する*function*は、起動すべき協調戦略関数についての情報であり、協調戦略関数に渡す引数を特に指定しない場合は戦略関数名のみ、引数を指定する場合は戦略関数名と引数とのリストになる。

receiveメソッド

エージェント間の協調処理の過程において、エージェント間の通信が必要になる場合がある。例えば他のエージェントからの要求に応じて協調処理を行う場合などである。多様な協調処理をメタレベル計算として実現するからには、この通信に付随する協調処理もメタレベル計算として実現する必要がある。そこでこのreceiveメソッドを問題レベルのプログラムを記述するプログラマに提供している。このメソッドには、受信したメッセージのクラスによって異なる処理を行なうdispatchというメソッドが用意されている。つまり問題レベルのプログラムでreceiveを利用してメッセージの受信を行えばdispatchメソッドとして記述した協調戦略を起動することができる。

協調戦略の記述法

キースロットの更新(参照)に伴って協調戦略関数を呼び出す場合、スロットオプション:strategy-setまたは:strategy-refを用いて指定された引数のほかに、エージェントクラ

スのインスタンス、キースロットのcarおよびcdrを引数として渡す。従って分離記述する協調戦略は、エージェントクラスのインスタンスを用いてエージェントの持つスロットにアクセスすることによって局所情報を集め、実行環境モデルを作成してキースロットのcdrに格納し、それを基に動作決定を行なうというアルゴリズムになる。また実行環境モデルの作成のみに限らず、協調戦略においてのみ用いる情報の管理も協調処理に含まれる。従って引数として渡されるキースロットのcdrが指すデータ構造には、管理すべき情報に対応する変数などを格納してもよい。

以上のようなインターフェースに従ってどのように協調戦略と問題レベルのプログラムとを記述すればよいかは次の章で協調戦略の例をあげて説明する。

5 システムを用いた協調処理の実現例

分散問題解決の処理効率に影響を及ぼす要因の1つとして、エージェント間またはエージェントとオブジェクトとの間の通信コストがある。エージェントの通信には問題解決上必要不可欠な通信と、むやみに行なうと全体の処理効率を下げうる通信とがある。前者の例としては各エージェントが持つ知識に強い一貫性を保つことが必要な共有資源が考えられる。後者の例としては、一貫性を保つことは必ずしも必要ないものの、処理効率を上げるためにはより新しい値を使った方が望ましいという性質を持つ知識が考えられる。このような実行中に生成(更新)される知識は、実行環境に応じて通信するかしないかを決定すべきである。このような知識は弱い一貫性を必要とする共有資源と考えられる。分散問題解決における資源は、局所的なもの、強い一貫性が必要なもの、弱い一貫性が必要なものの3種類に分類できるので、弱い一貫性保持の方法と強い一貫性保持の方法とを簡潔に記述できればプログラムの記述がより容易になるであろう。そこで本章ではこれら2つの協調戦略を利用者インターフェースに従って記述することにより、提案した記述方法ならびに実装したシステムの評価を行う。

本研究で以下のような協調戦略を、実装したシステムを用いて記述した場合の簡潔さについて考察してみた。

(1) 共有オブジェクトの一貫性保持に関する戦略

(2) 弱い一貫性を保持するための通信戦略

(1)は、複数のプロセスで共有する共有オブジェクトのコピーをどのプロセスに配置すべきかを決定するという戦略である。これは共有オブジェクトのコピーすべてに強い一貫性をもたせるという戦略の例である。一方(2)に示した通信戦略とは、どのような頻度で、どのエージェントと通信するのが効率的かを実行環境に応じて決定する戦略である。これは各エージェントの持つ知識に弱い一貫性を持たせることにより全体としての処理効率を上げるという戦略の例である。

5.1 共有オブジェクトの一貫性保持に関する戦略

分散環境において問題を解決してゆくためには共有オブジェクトあるいは知識の共有は不可欠である。強い一貫性が必要な共有オブジェクトに対して各エージェントがアクセスする場合、通信コストがかかる。このことを考慮すると、共有オブジェクトをどのような形でエージェントに共有させれば効率的かという問題が生じてくる[1,6]。これも分散問題解決で解決すべき問題であると考えると、この問題を解決するアルゴリズムも一種の協調戦略とすることができる。そこでこの共有オブジェクトに関する協調戦略を、実装したシステムを用いてメタレベル計算として実現することにした。

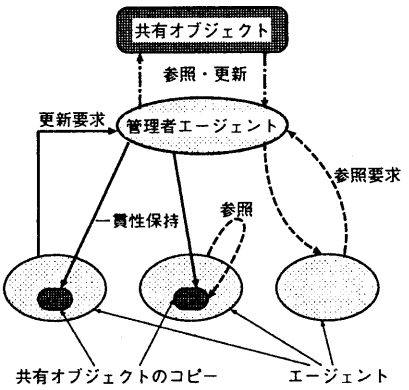

図6: 共有オブジェクトの一貫性保持

分散環境において、共有オブジェクトの参照や更新を行なうエージェントが多数存在する場合、参照回数の多いエージェントに対しては共有オブジェクトのコピーを持たせておけば参照の際の通信コストを削減することができる(図6)。しかし共有オブジェクトが更新された場合、コピーを持つすべてのエージェントに対してその旨を伝えなければならない。つまり共有オブジェクトのコピーの強い一貫性を保つ必要がある。今回取り扱ったのは、共有オブジェクトの更新が起こった時に、総エージェント数、現在コピーを持つエージェント数、通信コスト、そして各エージェント毎の参照回数を基に、どのエージェントに対してコピーを割り当てたら効率的かを決定する戦略である。従ってこの戦略で用いる変数の管理も戦略関数として分離記述することになる。共有オブジェクトへのアクセスやコピーの割り当てなどはすべて

メタレベル計算として実現するため、問題レベルのプログラムにおけるエージェントには普通のスロットへのアクセスと同じにしか見えない。

共有オブジェクトの管理を行なうエージェントの id を 0 とし、このエージェントを管理者エージェントと呼ぶ。各エージェントの持つ、共有オブジェクトに対応するスロット名を resource とすると、そのスロットの更新、参照によってどのような協調処理が起動されるかをまとめると以下のようになる。

(1)resource を参照した場合 もし共有オブジェクトのコピーを持たないならば、管理者エージェントに対して参照要求メッセージを送る。コピーを持つならばそれを直接参照する。

(2)resource を更新した場合 もし共有オブジェクトのコピーを持たないならば、管理者エージェントに対して更新要求メッセージを送る。もしコピーを持つならばそれまでのコピー参照回数をメッセージに追加して送る。

(3-1) 管理者エージェントが参照要求を受信した場合 その送信者の共有オブジェクト参照回数を記録し、共有オブジェクトの内容を返送する。

(3-2) 管理者エージェントが更新要求を受信した場合 総エージェント数、現在コピーを持つエージェント数、通信コスト、そして各エージェント毎の参照回数を基にしてどのエージェントにコピーを割り当てるかを決定し、コピーを持つすべてのエージェントに対して更新内容を放送する。

(1) と (2) の処理に対応する関数名をそれぞれ request-of-ref、update-copies とすると、これらはスロット resource の参照、更新に付随したメタレベル計算として起動させるので、以下のようにエージェントクラスを定義すればよい。

```
(defagent agent-with-shared-object ()
  ((resource            ...
                :strategy-ref request-of-ref
                :strategy-set update-copies)
   ...                                    ))
```

また (3-1) と (3-2) の処理は、メタレベルでの通信に付随する処理なので、メッセージのクラスに特定化した dispatch メソッドをそれぞれ定義すればよい。

以上より、共有オブジェクトの一貫性保持を実現する処理を問題レベルのプログラムから分離記述することができた。

今回本研究で取り扱ったのはすべての情報を 1 箇所 (管理者エージェント) に集めることによって共有オブジェクトを管理する戦略である。この他にも管理を分散させる戦略が考えられる。実装したシステムを用いてこのような戦略について研究することも今後の課題の 1 つである。

5.2 弱い一貫性を保持するための通信戦略

このような通信戦略が必要となるのは、例えば分枝限定法を用いて分散探索を行う場合である。分散探索の例として巡回セールスマン問題を解くエージェントの相互作用について見てみる (図 7)。各エージェントの性質は均一であり、分散して共有メモリ (黒板) から問題を取り出し、最短経路を探索してゆくものとする。探索中にエージェントは周囲のエージェントと、自分が獲得した経路の長さを通信によって交換しあう。この通信の目的は、探索領域の削減である。通信によってより短い経路を知れば不必要な探索を減らすことができ、全体の処理効率を上げることができる。しかし探索領域をあまり削減できないような通信は、通信コストを考慮すると全体の処理効率を下げる結果になるおそれがある。そこでエージェントは実行環境のモデルを作成し、それを参照して適切な通信対象数や通信頻度などを決定し、それに基づいて周囲のエージェントと通信を行なうことが必要になる。

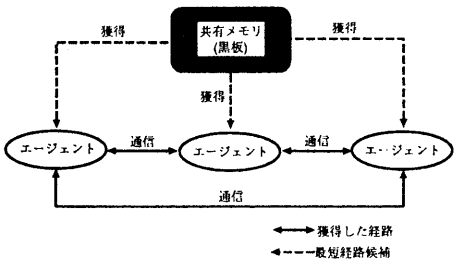

図 7: 巡回セールスマン問題におけるエージェントの相互作用

通信戦略の例として、「各エージェント自身が持っている 1 つの局所変数の更新履歴を実行環境モデルとして評価して通信対象を動的に決定する通信戦略」[11] について考察すると、これは探索解の更新率と通信対象数との履歴を基に通信対象を決定するという戦略である。つまりこれは探索解に対応するスロットの更新に伴って起動すべき戦略である。従ってこの戦略関数は、まずキースロットを基に実行環境モデルである更新履歴を作成し、これをキースロットの cdr に格納し、それをもとに通信対象を決定するというアルゴリズムになり、簡潔に分離記述することができる。

以下に通信戦略を用いた通信を行なうエージェントの一般的な動作を示す。

1. 然るべきタイミングで、局所情報を基に実行環境モデルを作成 (更新) する。
2. 実行環境のモデルを参照して、通信対象数や通信の頻度などを決定する。

3. この結果に基づいて周囲のエージェントと通信する。

あらかじめ記述した通信戦略に対応する関数名を **comm-st** とし、これを特定のスロット (**key-slot**) の更新に付随したメタレベル計算として起動させるためには、問題レベルのプログラムで以下のようにエージェントクラスを定義すればよい。

```
(defagent search-agent ()
    ((key-slot          ...
                :strategy-set comm-st)
     ...                                ))
```

この戦略はスロットの参照には無関係なのでスロットオプション:**strategy-ref** を用いる必要はない。

ある協調戦略に対して付加的な拡張を加えることより、さらなる効率化が期待できる場合がある。このような場合にもオブジェクト指向言語の性質より差分記述が可能なので戦略自体の拡張も容易に記述できる。

6 まとめと今後の課題

本稿では、まず現状の分散問題解決プログラムにおける協調処理の実現方法における問題点をとりあげ、その問題点を解消するような記述方法の必要性を指摘した。次に協調戦略について考察し、問題レベルのプログラムと協調戦略とを分離して記述するという方法を提案し、その枠組に基づいたシステムを、CLOS MOP を用いて実装した。そして最後に、エージェント全体の知識の一貫性保持に関する協調戦略2つを例に取り、実装したシステムを用いて記述した場合の簡潔さを示した。

本研究に関連する研究としては、エージェント指向言語システムである AgentSpeak[10] や、適応可能プログラミング言語システムに関する研究 [12] などが挙げられる。前者の AgentSpeak では周囲のエージェントからの要求に応じた協調処理の実現に重点が置かれている。しかし本研究では、エージェントの内部状態の変化に基づいた協調処理についても言及している。また我々は提案した枠組に基づいたシステムを、CLOS MOP を用いて実装し、協調戦略の分離記述を行なった。しかし [12] の研究で設計実装されている AL-1 という並行オブジェクト指向言語は、ベースオブジェクトの状態遷移に付随する振舞いをメタオブジェクトとして記述することを意図しており、これを利用して協調処理に関するエージェントの振舞いをメタオブジェクトとして定義することにより、同様に協調戦略の分離記述が可能であると思われる。

今後、今回実装したシステムを様々な協調戦略の研究に利用し、その一方で協調処理についての考察を重ねることにより、より汎用的な協調処理記述用言語の設計を行う予定である。

参考文献

[1] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming(PPoPP)*, pages 168–176, Seattle, 1990.

[2] Alan H. Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, 1988.

[3] Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. Coherent Cooperation Among Communicating Problem Solvers. *IEEE Transactions on Computers C-36*, pages 1275–1291, 1987. Reprinted in [2], pp.268–284.

[4] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[5] Victor Lesser and D. Corkill. Functionally accurate, cooperative distributed systems. In Bond and Gasser [2], chapter 4.3.1, pages 295–310.

[6] Kai Li and Paul hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Computing Systems*, 7(4):229–239, November 1989.

[7] Pattie Maes and Daniele Nardi, editors. *Meta-Level Architectures and Reflection*. 1988.

[8] Andreas Paepcke. User-Level Language Crafting: Introducing the CLOS Metaobject Protocol. In Andreas Paepcke, editor, *Object-Oriented Programming The CLOS Perspective*, chapter 3, pages 65–99. The MIT Press, 1993.

[9] Guy L. Steel Jr. et al. *Common Lisp the Language*. DEC press, second edition, 1990. 邦訳: 井田, "Common Lisp 第2版", 共立出版, 1991.

[10] Devindra Weerasooriya, Anand Rao, and Kotagiri Ramamohanarao. Design of concurrent agent-oriented language. In *Intelligent Agents*, pages 386–401. 1995.

[11] 楯崎 修二. 分散問題解決のための自律的構造選択戦略. In 奥乃 博, editor, マルチエージェントと協調計算 *III*, pages 177–184. 1994.

[12] 石川 裕. 適応可能プログラミング言語システムに関する研究. In 電子技術総合研究所彙法 第 *59* 巻 第 *3* 号, pages 33–40. 1995.